Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 215 (2008) 111–130
www.elsevier.com/locate/entcs

Multiple Concern Adaptation for Run-time Composition in Context-Aware Systems[1]

Dept. of Computer Science, University of Málaga, Spain.
Emails: jcamara@lcc.uma.es, canal@lcc.uma.es, salaun@lcc.uma.es

Abstract

Context-Aware computing studies the development of systems which exploit context information (*e.g.,* user location, network resources, time, etc.), which is specially relevant in mobile systems and pervasive computing. When these systems are built assembling pre-existing software components (COTS), the composition process must be able to solve potential interoperability problems, adapting component interfaces. In addition, the composition must be adapted to the execution conditions of such systems, which are likely to change at run-time, affecting component behaviour. This work presents an approach to the flexible composition of possibly mismatching behavioural interfaces in systems where context information can vary at run-time. Our approach enables composition at run-time, enabling dynamic changes in composition according to context changes. Furthermore, our approach simplifies the specification of composition/adaptation by keeping *Separation of Concerns*, and is able to handle context-triggered adaptation policies.

Keywords: Component-based Software Development, Run-time Composition, Model-based Adaptation, Separation of Concerns

1 Introduction

In recent years, software systems engineering has evolved from the development of applications from scratch, to the paradigm known as *Component-Based Software Development* (CBSD) [23], where third-party, pre-existing software components known as *Commercial-Off-The-Shelf* or COTS are selected and assembled in order to build fully working systems. The main advantage of

[1] This work has been partially supported by the project TIN2004-07943-C04-01 funded by the Spanish Ministry of Education and Science (MEC), and project P06-TIC-02250 funded by the Andalusian local Government.

doi:10.1016/j.entcs.2008.06.024

CBSD is that it promotes component reuse, saving time and money in system development. Due to the Black-box nature they exhibit, these components are equipped with public interfaces to access their functionality. However, most of the time a COTS component cannot be directly reused *as is*, requiring adaptation when composed with the rest of the system due to possible interoperability problems with other components.

Software Adaptation [6] is a field characterised by the modification of component interactions through the use of special components called *adaptors* [27], which are capable of enabling components with mismatching interfaces to interoperate. These are automatically built from an abstract specification of how mismatch can be solved (*i.e.* adaptation *mapping* or *contract*), based on the description of component interfaces. Particularly, recent research efforts [1,8,21,19] concentrate on behavioural interoperability, extending interfaces with a description of the protocol they follow, and ensuring correctness and termination of component interactions.

Traditionally, the context of reuse of a component used to be more or less static (*e.g.,* spreadsheets, banking systems, etc.), but the advent of mobile and pervasive computing has given rise to a whole new breed of systems where execution conditions are likely to change at run-time (*e.g.,* time, user location, resource availability, etc.). Although *Context-Aware computing* [9] has broadly studied the development of systems exploiting context information, it does not deal with the specificities of component-based systems. Component-based, context-aware systems must be able to reflect environmental changes affecting system behaviour, altering the composition at run-time.

This work advocates for the flexible (*i.e.,* modifiable at run-time) interaction between an arbitrary number of components depending on the current state of the execution of the system. This approach serves a double purpose: On one hand, it adapts the composition to the changing environmental conditions or *context* of the system. On the other hand, it works out the potential incompatibilities among components. This work develops and formalises the seminal ideas sketched in [4], extending previous work [5] which focuses on run-time composition and adaptation techniques. Run-time composition is an essential feature of our proposal since it avoids the costly generation of the full adaptor, and its recomputation when the system changes (*e.g.,* addition of a new service).

Moreover, we reduce the complexity of mapping specification by enabling separating concerns [12], breaking down the specification of adaptation into multiple context facets which express the different concerns which may affect system behaviour. Furthermore, our proposal is able to deal with adaptation policies which may depend on context changes (*i.e.,* context-triggered actions),

an important issue in Context-Aware computing which remains obviated by previous proposals in adaptation [10].

The rest of this paper is structured as follows: Section 2 presents a *Wireless Medical Information System* used to illustrate the different issues described in the remaining sections. Section 3 describes our run-time composition/adaptation proposal. Section 4 describes some related work. Finally, Section 5 draws up conclusions and further work.

2 Case Study: Wireless Medical Information System

In order to illustrate the different issues addressed in this paper, we describe a *Wireless Medical Information System* based on a real-world example, although simplified here for the sake of clarity. As it can be observed in Figure 1, the system consists of a client-server application which systematically processes the clinical information related to patients in a medical institution. There is a central server with a DBMS installed which is queried remotely from PDAs. Handheld devices and server are connected through a wireless network setup.

The client PDA must be able to work with three user profiles which have different privileges: while **Staff** can access a restricted set of information (*e.g.,* administrative info for attendants), **Doctors** and **Nurses** can access also medical information, and prescribe specific treatments for any given patient in the case of doctors. When a nurse applies a treatment previously prescribed by a doctor on a specific patient, the actions and/or medicines administrated must be entered in the application (treatment logging).

It is important to maintain the application operative on the PDA continuously, hence a lightweight DBMS component has been incorporated to each PDA, enabling the user to work locally whenever the wireless signal is lost (local mode). Moreover, since the storage on a PDA is very limited, only treatment prescriptions and logging are stored in the local DBMS. Patient information is retrieved from Radio Frequency Identification (RFID) [26] tags fixed on patient bracelets when in local mode. This is achieved through an RFID reader incorporated on each PDA. Every time the client on the PDA returns from local to remote mode, it is mandatory to synchronise the data stored locally with the central DBMS. This process must be automatically conducted by the application.

The client PDA is being reused from a legacy system which does not take into account user profiles, hence the appropriate restrictions must be applied at the composition level in order to limit the access rights to the DBMS as informally sketched above. Likewise, this client is built to work with a DBMS, independently of its location (the client does not know about the existence

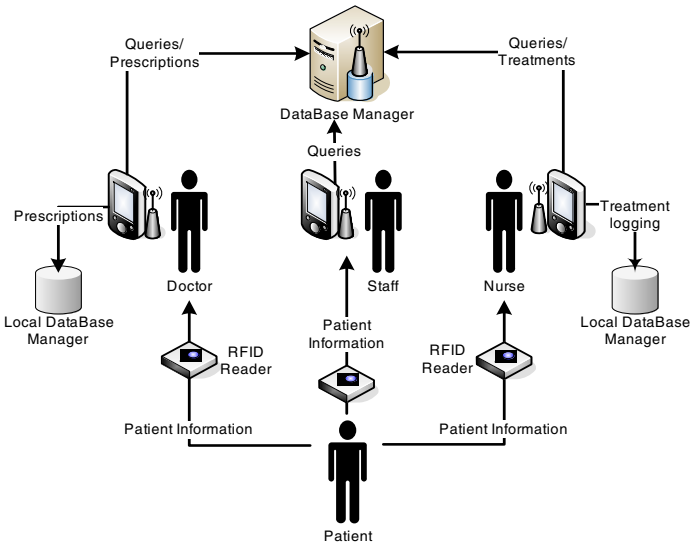

Fig. 1. *Wireless Medical Information System*

of the local DBMS nor the RFID reader), requiring adaptation to the new characteristics of the system.

3 Separation of Concerns for Run-time Composition and Adaptation

In this section, we first introduce our component and environment model. Second, we describe a graphical notation for our mapping which features separation of concerns, simplifying the specification of adaptation. Finally, we detail the process used for composition.

3.1 Component and Environment Model

Since this work deals with the behavioural interoperability level, we have to extend component interfaces with a description of the protocols they follow. In order to do so, we use *Labelled Transition System* (LTS) descriptions, which take the set of messages (both offered and required) in the signature of a component as input alphabet. Many automata-based languages can be used to describe behavioural interfaces (*e.g.,* Interface Automata, UML State Diagrams, etc.). We chose LTSs because of their simplicity and expressiveness, and also because they are widely used for design and formalisation purposes. In addition, this notation is particularly suited to be used as input for the algorithms presented in this work since it gives explicit information about the states of components. Moreover, it is user-friendly since its graphical repre-

sentation is straightforward, in contrast with other notations such as process algebras:

Definition 3.1 [LTS] A Labelled Transition System is a tuple (A, S, I, F, T) where: A is an alphabet formed by a set of events ($a!/a?$, emissions and receptions able to synchronise), S is a set of states, $I \in S$ is the initial state, $F \subseteq S$ are final states, and $T \subseteq S \times A \times S$ is the transition function.

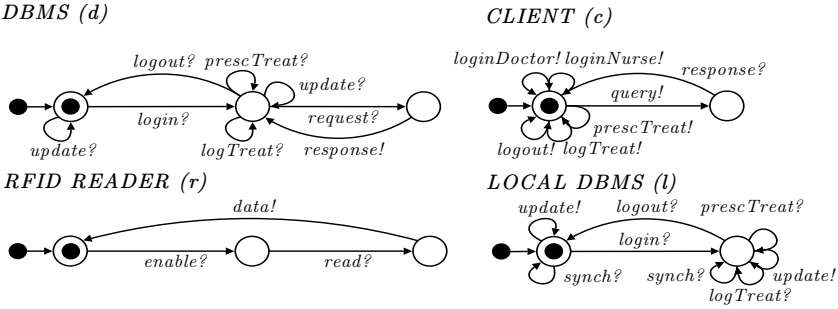

Fig. 2. Component protocols for the Wireless Medical Information System. Initial and final states are respectively noted in LTSs using bullet arrows and darkened states. Emissions and receptions are denoted by ! and ?, respectively.

Definition 3.2 [Execution Environment] An Execution Environment $E = \{e_1, \ldots, e_n\}$ is the set of environmental signals (or signals, for short) which do not belong to any particular component.

Example 3.3 Figure 2 depicts the different protocols for the components in our case study: The *CLIENT* component can log an user in/out (*loginDoctor*!/*loginNurse*!/*logout*!), request the insertion of a given treatment on the database (*prescTreat*!), log the administration of a treatment (*logTreat*!), or request some information to the server *query*!, returned by *response*?. It is worth noticing that the client grants the same privileges to all users. The *DBMS* and *LOCAL DBMS* components have analogous actions, with the exception that the latter only accepts *prescTreat*?/*logTreat*? requests, and *synch*?, which triggers a synchronisation process with the DBMS components. This synchronisation is effectively performed by *update*!/*update*? on both component interfaces. Finally, the *RFID READER* component first has to be enabled (*enable*?). Subsequently, it can receive a *read*? command, returning the requested information on *data*!. In our case study, when the wireless signal is found or lost, we will consider the pair of signals $E = \{connected!, disconnected!\}$ for our execution environment.

The composition in this system must take into account a couple of different concerns: **(i)** User profile. The client we are reusing does not distinguish user privileges, therefore we must provide the means to restrict user privileges based

on user profiles (*e.g.*, a *prescTreat*! should not be issued to the DBMS unless a doctor user is logged in -*loginDoctor*!-). **(ii)** Wireless coverage. Working in connected mode, queries are issued to the DBMS, but when in local mode, a *query*! request must be issued to the *RFID READER* component.

Independently of the different concerns to be considered for the composition, there are interoperability issues to be solved relative to the different interfaces of our case study:

- Name mismatch occurs when a particular process is expecting a particular input event, and receives one with a different name (*e.g.*, *CLIENT* sends *query*! while *DBMS* is expecting *request*?).
- 1-to-many interaction is given if one or more events on a particular interface have not an equivalent in the counterpart's interface. If we take a closer look at the *CLIENT* and *RFID READER* interfaces, it can be observed that while the client is just sending *query*! when it wants to read some data, the reader is expecting two messages (*enable*? and *read*?). While the latter actually requests the data to the reader, the former has no correspondence on the *CLIENT* interface. Hence, the composition process has to solve this mismatch by making the reader evolve independently, through the reception of *enable*? before each *read*? request.

Finally, composition must also consider the synchronisation of local and remote DBMS, triggering *synch*? when the system recovers connectivity (*connected*!).

3.2 Mapping

A mapping establishes correspondences or bindings between operations on the different component signatures in order to make interactions explicit and solve possible mismatch between them. When this correspondence is fixed or static, the specification of the mapping is relatively simple [2], but it gets more complicated in systems which require the modification of these correspondences at run-time depending on the current state of context. Moreover, there is an additional complexity in the specification of changes in the context derived from all the possible combinations of factors which may influence the execution of the system. In order to tackle this problem, the mapping described in this work features Separation of Concerns, a divide-and-conquer strategy which makes the problem easier to manage by breaking it down into pieces.

To begin with, correspondences are specified in our mapping by denoting communication among several components and the environment. For that we use *synchronisation vectors* [7]:

Definition 3.4 [Synchronisation Vector] A synchronisation vector (or vector for short) for a set of components $C_{i\in\{1,..,n\}} = (A_i, S_i, I_i, F_i, T_i)$, and an execution environment E is a tuple $\langle l_1, \ldots, l_n, e_1, \ldots, e_m\rangle$ with $l_i \in A_i \cup \{\varepsilon\}$, $e_{j\in\{1,..,m\}} \in E \cup \{\varepsilon\}$, where ε means that a particular component or signal does not participate in a synchronisation.

A vector may involve any number of components and/or signals and does not require interactions on the same names of events as it is the case in process algebras [16,15]. To identify component messages in a vector, their names are prefixed by the component identifier ($\langle c : prescTreat!, l : prescTreat?\rangle$), whereas signals are not prefixed, *e.g.,* $\langle connected!, l : synch?\rangle$. Moreover, in a vector, all the components which do not participate in an interaction may be removed to simplify the notation.

Communication expressed by vectors affects the state of context. To keep track of changes in context we use *vector expressions*. These are predicates over vectors, meaning that given a specific vector, a vector expression can either match it or not:

Definition 3.5 [Vector Expression] A vector expression is a tuple $\langle l_1, \ldots, l_n\rangle$ where each $l_{i\in\{1,..,n\}}$ is an expression designating one or several events/signals. Expressions may contain the following wildcards:

- ***** designates a sequence of 0 or more characters to be used either on the event prefix or identifier. For instance, $*:login?$ designates in our case study both $d:login?$ and $l:login?$.
- **..** designates 0 or more events/signals. Hence, a vector expression such as $\langle connected!, ..\rangle$ would match on $\langle connected!\rangle$ or $\langle connected!, l:synch?\rangle$, for instance.

We describe a mapping through an incremental specification, focusing on the different facets or concerns involved in the composition. Each concern is represented as a *context facet*, where the changes between the different states of that particular context facet are triggered either by component messages or signals designated by vector expressions. Specifically, when an expression matches on the vector which is currently being applied in the composition, the transition is triggered. The order of events and signals does not need to coincide both in the expression and the vector for them to match.

Definition 3.6 [Context LTS] A Context LTS for a set of components $C_{i\in\{1,..,n\}} = (A_i, S_i, I_i, F_i, T_i)$, and an execution environment E is a tuple $(A_c, S_c, I_c, F_c, T_c)$ specified over a set of vectors V where: A_c is an alphabet (set of vector expressions specified over A_i and E), $S_c \subseteq Id \times V$ is a set of context states, $I_c \in S_c$ is the initial state, $F_c \subseteq S_c$ are final states, and

$T_c \subseteq S_c \times A_c \times S_c$ is the transition function. *Id* stands for a set of identifiers.

Definition 3.7 [Context Facet] A Context Facet is is a tuple $(CLTS, Vc)$ where:

- $CLTS$ is a context LTS defined for a given set of components and an execution environment.
- Vc is a set of vectors used by $CLTS$ context states.

Vectors will only be taken into account for the composition when associated to the facet's current state. In addition, the mapping may contain a set of *global* vectors which are not associated to any particular context facet and are always considered for composition. Moreover, vectors in different facets may share the same identifier (in such a case we refer to them as *vector declarations*). This characteristic is used to be able to specify a modification over a part of the behaviour of the system which has already been specified by overriding it. Facets have a precedence order assigned, hence the declaration of a vector in a facet with higher precedence overrides a lowest precedence declaration. All facets have a different precedence order higher than 0. Global vectors have a precedence order $p = 0$, and may be overriden by vectors on facets.

Definition 3.8 [Mapping] A context mapping built over components $C_{i\in\{1,\ldots,n\}} = (A_i, S_i, I_i, F_i, T_i)$, is defined as a tuple $(CF_{j\in\{1,\ldots,m\}}, Vg, P)$ where:

- $CF_{j\in\{1..m\}} = (CLTS_j, Vc_j)$ is a set of context facets.
- Vg is a set of vectors global to all context facets and states in the mapping.
- and $P = \{p_1, \ldots, p_m\}, p_j \in \{1, \ldots, m\}, \forall\, p_k, p_l \in P\ p_k \neq p_l \wedge p_j > 0$ is a list of precedence orders, one for each context facet.

It is worth noticing that there may be cases in which we may need to assign a specific precedence order to just one vector. This could be expressed by adding the precedence order in the vector declaration (*e.g.*, $v_{conn}[3] = \langle\ connected!, l : synch?\ \rangle$).

Example 3.9 Figure 3 depicts context facets for the mapping we use for our case study. These form the mapping along with the precedence orders $P = \{p_{WC} = 1, p_{UP} = 2\}$, and the following set of global vectors Vg:

$$v_{lid} = \langle c : loginDoctor!, d : login?\rangle \qquad v_{resp} = \langle c : response?, d : response!\rangle$$

$$v_{lin} = \langle c : loginNurse!, d : login?\rangle \qquad v_{lo} = \langle c : logout!, d : logout?\rangle$$

$$v_{ptr} = \langle c : prescTreat!, d : prescTreat?\rangle \qquad v_{qry} = \langle c : query!, d : request?\rangle$$

$$v_{ltr} = \langle c : logTreat!, d : logTreat?\rangle \qquad v_{up} = \langle l : update!, d : update?\rangle$$

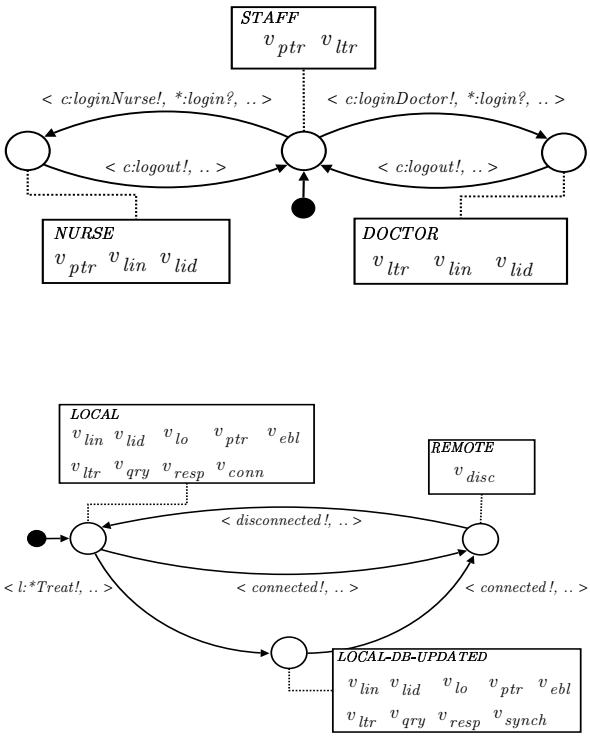

Fig. 3. Mapping facets for the case study.

It can be observed that while the set of global vectors specifies the general behaviour of the system, the different facets modify composition according to the part of the context they are concerned about:

- *WIRELESS COVERAGE* (*WC*) is concerned about the local/remote operation mode. This facet specifies alternatives for the global vectors in the following way:
  - When the state of this facet is *LOCAL*, treatment prescription and logging are performed on the local DBMS (vectors v_{ptr} and v_{ltr}, respectively). Likewise, queries for patient data are performed on the RFID reader (vectors v_{qry}, v_{ebl} and v_{resp}). Notice that v_{ebl} will help to solve the 1-to-many interaction problem described in our case study, by making the RFID reader evolve independently whenever it is ready to receive *enable*?.
  - The state $LOCAL - DB - UPDATED$ is similar to *LOCAL*, although it is entered when the local database is modified. In addition to the vectors described for local mode, v_{synch} is included for local-remote DBMS

synchronisation. This state is introduced in the mapping in order to avoid unnecessary synchronisations (when the local database has not been updated) which can cause additional network traffic.

- *USER PROFILE* (*UP*) modifies the functionality of the system according to the current user profile: v_{ptr} restricts treatment prescription in this facet (this vector appears in all states except *DOCTOR*, which are entitled to enter prescriptions). Treatment logging is similarly restricted for non-nurses by v_{ltr}.

3.3 Composition

Once we have described the inputs to our approach, we will detail the process followed for composition and adaptation. First, we illustrate the selection of active vectors for a particular state of the global context (*i.e.,* the combination of the active states of all context facets). Second, we sketch the approach used to ensure correct termination of the system. Finally, we describe the composition algorithm, illustrating it with an execution trace coming from our case study.

3.3.1 Selection of an Active Vector Set

For each state of the global context, there is a vector set that describes the possible interactions among the components. In order to select those vectors, namely *active vectors*, we define the function *active*, which takes as input mapping $M = (CF_{j\in\{1,\ldots,n\}}, Vg, P)$ and the list of current states for the facets *cstates*. It returns the set of active vectors for the current state of the global context (including global vectors which have not been overriden). In order to do that it makes use of the $\uplus$ operator, which returns the set of couples (v, p) of the different sets given as input, where p is the highest precedence for v. For instance, for the sets $A = \{(v_1, 1), (v_2, 1), (v_3, 2)\}$ and $B = \{(v_1, 2), (v_2, 0)\}$, $A \uplus B = \{(v_1, 2), (v_2, 1), (v_3, 2)\}$. Function id returns the identifier of context state s.

$$active(M, cstates) =$$
$$\{v \mid (v,p) \in \{(v,0) \mid v \in Vg\} \uplus \{(v, P[j]) \mid v \in V,\ v \in Vc_{j\in\{1..n\}},\ \exists\, s \in S_{cj}\ s = cstates[j] \wedge (id(s), V) \in s\}\}$$

$$A \uplus B = \{(v,p) \mid (v,p) \in A \cup B \wedge \forall (v',p') \in (A \cup B) \backslash \{(v,p)\}, v = v', p \geq p'\}$$

Example 3.10 In order to illustrate how active vectors are selected for a given state of the context, we use the mapping for our case study depicted in Figure 3. We focus on the particular vector v_{ltr}, declared as $v_{ltr} = \langle c :$

$logTreat!, d:logTreat?\rangle$ in the global set of vectors (the $logTreat!$ message is issued to the remote DBMS). v_{ltr} is defined as $v_{ltr} = \langle c:logTreat!, l:logTreat?\rangle$ in the *WIRELESS COVERAGE* facet (the operation is performed on the local DBMS), and as $v_{ltr} = \langle c:logTreat!\rangle$ in *USER PROFILE* (the operation is not performed, since the client request $logTreat!$ corresponds to no action on the rest of the components). We also consider that the set of current states in facets are $C = \{DOCTOR, LOCAL\}$. Focusing on *WIRELESS COVERAGE*, we can observe that since v_{ltr} is associated to the *LOCAL* state, the declaration on this facet overrides the global declaration. Similarly, since v_{ltr} is associated to *DOCTOR* and the precedence of the *USER PROFILE* is higher, the currently dominant declaration is again overriden. Finally, the operation is not performed since the prevailing declaration is $v_{ltr} = \langle c:logTreat!\rangle$. This is consistent with our example since doctors are not allowed to enter administrated treatments on the application. To sum up, we keep the vector with the highest precedence in case there are several vectors identified similarly. The complete set of active vectors for this particular state of the global context is:

$$v_{lin} = \langle c:loginNurse!\rangle \quad v_{resp} = \langle c:response?, r:data!\rangle \quad v_{lo} = \langle c:logout!, l:logout?\rangle$$
$$v_{lid} = \langle c:loginDoctor!\rangle \quad v_{qry} = \langle c:query!, r:read?\rangle \quad v_{ltr} = \langle c:logTreat!\rangle$$
$$v_{conn} = \langle connected!\rangle \quad v_{ebl} = \langle r:enable?\rangle \quad v_{up} = \langle l:update!, l:update?\rangle$$
$$v_{ptr} = \langle c:prescTreat!, l:prescTreat?\rangle$$

3.3.2 Ensuring Correct Termination

Adapting interfaces at the protocol level implies not engaging the system into deadlocking executions. A deadlock state is a state which is not final and in which a process cannot evolve. A system deadlocks when all its constituent components are blocked because at least one of them is in a deadlock state. Since deadlock removal cannot be performed before the application of the adaptor as in approaches which generate full adaptor descriptions [2,10], we have to ensure the existence of one termination state for the system before every application of a vector. Hence, if we cannot find a sequence of vectors to be applied leading to a global termination state for the system after the application of vector v, we do not apply that vector.

The vector to be applied at an specific moment is selected from a set of *applicable vectors* (*i.e.,* active vectors which in addition can make the system evolve in a given moment). Function $applicable_V$ returns the set of applicable vectors from a set of vectors V for the list of current *states* associated to components C_i. Note that for our purposes, we will select applicable vectors

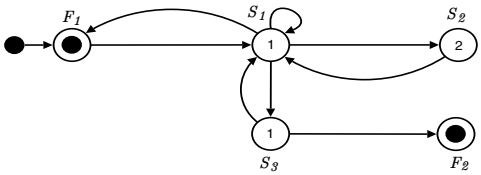

Fig. 4. Example of LTS tagged with minimum distances to final states.

from the set of currently active vectors:

$$applicable_V(states, C_i, V, E) = \{v \mid v \in V, (\forall\, l \in v)\ ((s_{j\in\{1..n\}} \in S_j, s_j = states[j],\ (s_j, l, s'_j) \in T_j,\ l \neq \varepsilon) \vee (l \in E))\}$$

In order to keep track on the evolution of the system, function *next_states* computes the states of the components involved in the composition after the application of an specific vector in the current state of the components:

$$next\_states(\langle l_1, \ldots, l_n\rangle, states, T_i, E) = [s'_1, \ldots, s'_n]$$
$$\text{where } \forall\, i \in \{1, \ldots, n\}\ \exists (s_i, l_i, s'_i) \in T_i,\ s_i = states[i], l_i \neq \varepsilon, l_i \notin E$$

In order to know if the search process has found a goal state (*i.e.,* a global termination state), we check if all the components have reached their final state. We define the function *final* as:

$$final(states, F_i) = states[1] \in F_1 \wedge \ldots \wedge states[n] \in F_n$$

Considering the nature of our search problem, the use of an informed search algorithm looks like a good strategy in order to find potential solutions efficiently. Specifically, we make use of the A* algorithm [14], a particular best-first search strategy which determines the mimimum cost path from a given node n to a goal state by expanding the most promising candidate paths first. However, we have to provide guidance information for this search. This is achieved by defining a heuristic estimation function $h(n)$ of the cost of arriving from the current state of the components to a global termination state in the composition. In order to determine the heuristic estimation to be used:

- We define the minimum distance from a specific state s in a component LTS C to a final state $d(s, C)$, as the minimum number of events which have to be applied to traverse the LTS up to a final state. Figure 4 depicts a sample LTS with states tagged with minimum distances to final states. This distance is computed using a variant of the Floyd-Warshall algorithm [18].
- Given a set of components, $C_{i\in\{1,\ldots,n\}} = (A_i, S_i, I_i, F_i, T_i)$, a set of current states $states, i \in \{1, \ldots, n\}, states[i] \in S_i$, we define the minimum global distance to a final state for the whole system as:

$$D(states, C_i) = \textstyle\sum_{i=1..n} d(states[i], C_i)$$

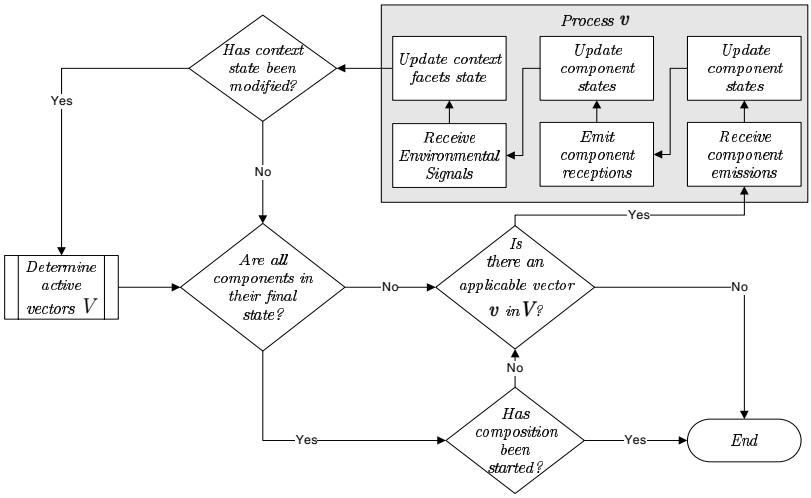

Fig. 5. Composition process.

- The heuristic used to inform about how good the application of a specific vector v is, can be expressed as $h(v, states, C_i) = D(next\_states(v, states, T_i), C_i)$

The heuristic estimate $h(n)$ is admissible in our case (*i.e.,* it never overestimates the actual cost from node n to a goal). This is because the distance function d that we have defined always returns the minimum of the distances from a state in the LTS to a final state, resulting in a lower bound of the estimation. This admissibility guarantees A* to return an optimal solution, if one exists [11].

3.3.3 Composition process

Figure 5 sketches the run-time composition process we propose (for the formal definition of the composition process refer to Appendix A):

(i) The set of active vectors dependent on the current states of the different facets of the context is selected. This selection is performed as described in 3.3.1.

(ii) Run-time composition should avoid to engage into execution branches that may lead to deadlock situations. At this stage the state of the components is checked, and if all of them are in a final state, the composition finishes. Otherwise, the composition engine attempts to select an applicable vector v which may lead to a global correct termination state for the system. If such vector does not exist, the composition process ends as well.

(iii) Vector v is processed. First, the engine receives the emissions specified in v. Notice that the engine operates reversing the direction of communication with respect to the events specified in vectors. Next, the engine sends the receptions specified in v. After processing both emissions and receptions the state of the components is updated accordingly. Finally, environmental signals are received by the engine as well, and the state of context facets is updated by matching vector expressions on context facet LTSs with the vector being currently applied (v).

(iv) Finally, if the state of the global context has changed, the set of active vectors is updated according to the new state of the context, and composition continues.

Example 3.11 In order to illustrate the composition process, we describe in Figure 6 a sample execution trace for the composition engine in our case study:

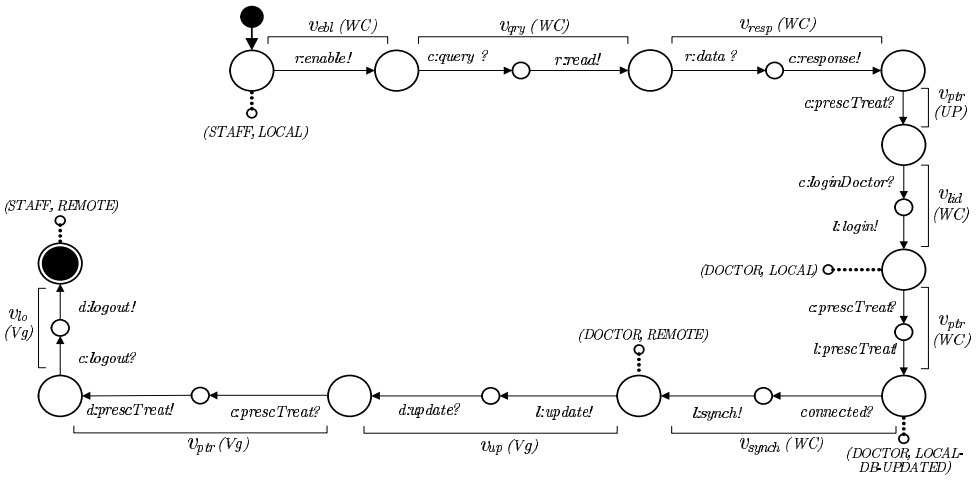

Fig. 6. Sample event trace for the case study. Applied vectors are tagged with the facet they belong to.

- The initial state of the global context is $C = \{STAFF, LOCAL\}$. The *RFID READER* is waiting to be enabled, so the composition engine applies $v_{ebl}(WC)$, making the component evolve independently. Next the client makes a query for patient data ($c : query!$), and the engine applies $v_{qry}(WC)$, and $v_{resp}(WC)$ subsequently as data is returned.
- The client requests a treatment prescription, which is received by the composition engine, but it is obviated since the current user profile is not allowed to perform that operation. Hence, $v_{ptr}(UP)$ is applied.
- The user logs in as doctor on the local DBMS through the application of

$v_{lid}(WC)$. This causes a change in the state of context facet UP. The new state of the global context is $C = \{DOCTOR, LOCAL\}$ and active vectors are selected again. The client requests a new treatment prescription, which in this case is effectively performed through the application of $v_{ptr}(WC)$. The update of the local DBMS causes a change in the state of context facet WC. The new state of the global context is now $C = \{DOCTOR, LOCAL-DB-UPDATED\}$.

- The PDA recovers the wireless network signal at this stage. This causes the application of vector $v_{synch}(WC)$, which triggers the synchronisation of local and remote DBMS. The new state of the global context is now $C = \{DOCTOR, REMOTE\}$. Subsequently, the $LOCAL\ DBMS$ causes the application of v_{up} in order to perform the effective synchronisation process.
- The client requests a new treatment prescription, which in this case is performed on the remote DBMS, since coverage is now available. This is achieved through the application of $v_{ptr}(Vg)$.
- Finally, the user logs out, $v_{lo}(Vg)$ is applied and the composition finishes correctly.

4 Related Work

In Context-Aware Computing applications can discover and take advantage of contextual information (such as user location, time, resource availability, etc.). Although this topic has been broadly studied and the usefulness of this technology has been demonstrated [9], this paradigm does not explicitly deal with the composition and adaptation of software entities within the system.

Regarding separation of concerns, different proposals in the field of Aspect-Oriented Programming have been put forward related to the adaptation of applications. For instance, in [24], Tanter *et al.* supply support for an aspect language with constructs which adequate the behaviour of aspects to the state of context. The notion of context supported refers to a set of attributes or variables in the application and their value at some specific point in time (context snapshots). Vanderperren *et al.* present in [25] an extension for the JAsCo programming language [22] which allows the triggering of aspects describing their applicability in terms of a sequence or protocol of previously matched run-time events. These approaches do not deal with the different issues related to the composition of entities (including interface adaptation), providing only a way to extend aspect behaviour according to context information in a static way.

In [17], Mukhija and Glinz describe a contract-based adaptive software architecture which deals with the adaptation of applications at run-time ac-

cording to their execution environment. While this approach supports the notion of composition, it does not deal with the different issues related to protocol adaptation. For instance, if a component is going to be replaced by a new version, both must conform to the same interface. Moreover, contracts define alternative configurations for the composition according to different states of the context. Likewise, Braione and Picco [3] propose a calculus to specify contextual reactive systems separating the description of behaviours and the definition of contexts in which some actions are enabled or inhibited.

Our approach goes beyond [17] and [3] by allowing a separate representation of the different concerns involved in the composition, which is automatically handled by the composition engine. Moreover, our proposal takes contextual information into account while integrating components with mismatching interfaces, since vectors defined in our mapping notation can work behavioural mismatch out.

Cubo *et al.* describe in [10] an adaptor-based approach to context-aware adaptation. However, the state of context depends exclusively on the exchange of messages among components during execution. Hence, while this proposal can work out behavioural mismatch situations, adaptation policies depending on other type of context information (*i.e.,* environment) is not supported. Compared to our proposal, [10] does not support separation of concerns. As a consequence, every possible state of the context has to be manually specified by the developer writing the mapping, increasing the complexity of its specification. Moreover, the adaptor generation process does need to consider every possible state of the system (not only context). This implies that the adaptor is no longer valid if new context information or components are added or removed at run-time, requiring the costly generation of a new adaptor. On the contrary, our approach does not require any recomputation in case of changes to the system (context information or components), since composition and adaptation are generated and conveniently modified according to the description given in the mapping at run-time.

5 Conclusions

In this paper we have presented an approach to the composition and adaptation of mismatching components in systems where its behaviour may be affected by the execution environment. Our approach applies composition at run-time rather than generating a full adaptor off-line, and simplifies the specification of adaptation applying separation of concerns to the specification of the adaptation mapping. The proposed approach adapts the composition to the changing context of the system and works out potential incompatibilities

among components, while taking into account context-triggered actions.

Regarding future work, a first perspective is reconfiguration of the system. While the nature of the mapping and the compositional process we have presented enables the transparent modification of the system, this work does not currently deal with the specifics of the reconfiguration process which takes place after the addition or removal of new context information or components as the system is running. Mapping or component update must be performed only at specific safe points, since the modification of this information at any other point could harm the correct execution of the system. The same applies to context changes during already running transactions, which should be able to execute correctly. A potential solution to this problem is delimiting the boundaries of transactions and delaying the application of context changes until they end.

Our main perspective is to implement this proposal as a composition engine, using *Aspect-Oriented Programming* (AOP) [13], where unlike in traditional platforms and languages, a particular system can be modified without altering its (base) code. This is achieved by separately specifying modifications as *aspects*, and a description of their relation with the current system. Then the AOP environment *weaves* or composes aspects and base code into a coherent program. This weaving process can be performed at different stages of the development, ranging from compilation-time to run-time [20] (dynamic weaving). In this dynamic approach, the virtual machine or interpreter running the code is aware of aspects and controls the weaving process. Hence, aspects can be applied and removed at run-time in a transparent way. Dynamic AOP will enable us to shape up the composition engine as aspects able to: (i) intercept communication (*i.e.,* service invocations) between components; (ii) apply the composition process introduced in this proposal *wrt.* the adaptation mapping in order to make the right message substitutions; (iii) forward the substituted messages to their recipients transparently.

References

[1] A. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), 1997.

[2] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A Formal Approach to Component Adaptation. *The Journal of Systems and Software*, 74(1), 2005.

[3] P. Braione and G.P. Picco. On Calculi for Context-Aware Coordination. In *Proc. of COORDINATION'04*, volume 2949 of *LNCS*. Springer, 2004.

[4] Javier Cámara, Gwen Salaün, and Carlos Canal. On run-time adaptation in context-aware systems. In *Proc. of M-ADAPT'07*, 2007. (in press).

[5] Javier Cámara, Gwen Salaün, and Carlos Canal. Run-time Composition and Adaptation of Mismatching Behavioural Transactions. In *Proc. of SEFM'07*. IEEE Computer Society Press, 2007. (in press).

[6] C. Canal, J.M. Murillo, and P. Poizat. Software Adaptation. *L'Objet*, 12(1), 2006. Special Issue on WCAT'04.

[7] C. Canal, P. Poizat, and G Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *Proc. of FMOODS'06*, volume 4037 of *LNCS*. Springer, 2006.

[8] Carlos Canal, Lidia Fuentes, Ernesto Pimentel, José M. Troya, and Antonio Vallecillo. Adding Roles to CORBA Objects. *IEEE Transactions on Software Engineering*, 29(3), March 2003.

[9] G. Chen and D. Kotz. A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000-381, Dartmouth College, 2000.

[10] J. Cubo, G. Salaün, J. Cámara, C. Canal, and E. Pimentel. Context-Based Adaptation of Component Behavioural Interfaces. In *Proc. of COORDINATION'07*, volume 4467 of *LNCS*. Springer, 2007.

[11] R. Dechter and J. Pearl. Generalized Best First Search Strategies and the Optimality of A*. *J. ACM*, 32(3), 1985.

[12] E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982.

[13] R. Filman and D. P. Friedman. *Aspect-Oriented Software Development*, chapter Aspect-Oriented Programming Is Quantification and Obliviousness. Adisson-Wesley, 2005.

[14] P. Hart, N. Nilsson, B., and Raphael. A Formal Basis for the Heuristic Determination of Minimum-Cost Paths. *IEEE Trans. on Systems Science and Cybernetics*, SSC-4(2), 1968.

[15] ISO. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, International Standards Organisation, 1989.

[16] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[17] Arun Mukhija and Martin Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *Proc. of ARCS'05*, volume 3432 of *LNCS*. Springer, 2005.

[18] Christos H. Papadimitriou and Martha Sideri. On the floyd-warshall algorithm for logic programs. *J. Log. Program*, 41(1), 1999.

[19] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Software Eng*, 28(11), 2002.

[20] A. Frei Popovici A. and G. Alonso. A Proactive Middleware Platform for Mobile Computing. In *In Proc. of Middleware'03*, LNCS. Springer, 2003.

[21] Ralf H. Reussner. Automatic component protocol adaptation with the coconut tool suite. *Future Generation Computer Systems*, 19(5), 2003.

[22] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: an Aspect-Oriented Approach Tailored for Component Based Software Development. In *AOSD*, 2003.

[23] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Adisson-Wesley, 2003.

[24] Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. In *Proc. of SC'06 at ETAPS'06*, volume 4089 of *LNCS*. Springer, 2006.

[25] Wim Vanderperren, Davy Suvée, María Agustina Cibrán, and Bruno De Fraine. Stateful Aspects in JAsCo. In *Proc. of SC'05*, volume 3628 of *LNCS*. Springer, 2005.

[26] Roy Want. Enabling ubiquitous sensing with RFID. *IEEE Computer*, 37(4), 2004.

[27] Daniel M. Yellin and Robert E. Strom. Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2), March 1997.

A Composition Algorithm

Function *select_vector* returns a single applicable vector non-deterministically chosen from V, whose application can lead to a final state for the whole system (function *exist_final* corresponds to the search process described in 3.3.2, returning *true* if a correct global termination state exists for the system after the application of v):

$$select\_vector(states, C_i, V, E) = \begin{cases} v & \text{if } v \in applicable_V(states, C_i, V, E) \neq \varnothing \\ & \wedge\ exist\_final(states, C_i, V, v) \\ v_\perp & \text{otherwise (no vector applicable)} \end{cases}$$

We define the functions *emissions*, *receptions*, and *signals* which return the set of emissions, receptions, and signals respectively, of any given synchronisation vector. Note that functions *emissions* and *signals* take an execution environment E as input in order to discriminate actual signals from component emissions:

$$\begin{gathered} emissions(\langle l_1, \ldots, l_n \rangle, E) = \{e \mid l_i = e! \wedge e! \notin E\} \\ receptions(\langle l_1, \ldots, l_n \rangle) = \{r \mid l_i = r?\} \\ signals(\langle l_1, \ldots, l_n \rangle, E) = \{s \mid l_i = s! \wedge s! \in E\} \end{gathered}$$

The function $match(v, ve)$ returns true if the supplied vector expression ve matches with vector v, being used to update the state of the different context facets according to the vector which is currently being applied within the composition process.

This algorithm is an extension of the work described in [5], where further details can be found about its correctness, termination and prototype implementation.

Algorithm 1 *runtime_composition*

Composes a set of components at run-time wrt. a context mapping and an execution environment

inputs components $C_{i\in\{1,\ldots,n\}} = (A_i, S_i, I_i, F_i, T_i)$, context mapping $(CF_{j\in\{1,\ldots,m\}}, Vg, P)$, with $CF_j = (CLTS_j, Vc_j)$), execution environment E

1: $states := [I_1, \ldots, I_n]$
2: $cstates := [I_{c1}, \ldots, I_{cn}]$
3: $started := false$ // *composition start condition*
4: $CVS := active(CF_j, cstates, P, Vg)$
5: $v := select\_vector(states, C_i, CVS, E)$
6: **while** $v \neq v_\perp \wedge (\neg final(states, F_i) \vee \neg started)$ **do**
7: $started := true$
8: $Rec := emissions(v, E)$
9: $Em := receptions(v)$
10: $Sg := signals(v, E)$
11: **repeat** {receptions}
12: $r? \mid r \in Rec, j \in \{1, \ldots, n\}, s_j = states[j], (s_j, r!, s'_j) \in T_j$
13: $states[j] := s'_j$ // *update of component states*
14: $Rec := Rec \backslash \{r\}$
15: **until** $Rec = \varnothing$
16: **repeat** {emissions}
17: $e! \mid e \in Em, j \in \{1, \ldots, n\}, s_j = states[j], (s_j, e?, s'_j) \in T_j$
18: $states[j] := s'_j$ // *update of component states*
19: $Em := Em \backslash \{e\}$
20: **until** $Em = \varnothing$
21: **repeat** {signals}
22: $signal? \mid signal \in Sg$
23: $Sg := Sg \backslash \{signal\}$
24: **until** $Sg = \varnothing$
25: **for all** $j \in \{1, \ldots, m\}, s_{cj} = cstates[j], (s_{cj}, ve, s'_{cj}) \in T_{cj}$ **do**
26: **if** $match(v, ve)$ **then**
27: $cstates[j] := s'_{cj}$ //*update context facet states*
28: **end if**
29: **end for**
30: $CVS := active(CF_j, cstates, P, Vg)$
31: $v := select\_vector(states, C_i, CVS, E)$
32: **end while**